interChange

Newsletter of the International SGML/XML Users' Group **http://www.isgmlug.org/**

In the News . . .

XLink an Approved W3C Standard at Last

The *XML Linking Language (XLink) Version 1.0* finally became an approved W3C Recommendation in June. XLink provides a framework for creating both basic unidirectional links and more complex linking structures by allowing XML documents to assert linking relationships among more than two resources, associating metadata with a link and storing links in a location separate from the linked resources. On the same day the associated *XML Base* specification, which provides facilities for defining base URIs for parts of XML documents, also became an approved W3C Recommendation. Hopefully, these two standards will shortly find their way into all the major Web browsers.
XLink Specification:
http://www.w3.org/TR/xlink/
XBase Specification:
http://www.w3.org/TR/xmlbase/

SMIL Now Animated

The *Synchronized Multimedia Integration Language (SMIL 2.0)* Specification became a W3C Proposed Recommendation in June. SMIL 2.0 defines an XML-based language that allows authors to write interactive multimedia presentations and provides a syntax and semantics that can be used in other XML-based languages that need to represent timing and synchronization. In the draft, the existing functionality has been modularized and new functionality has been added, especially in the area of animation effects. *SMIL Animation* provides a specification of animation functionality for use within XML documents. The specification describes an animation framework as well as a set of XML animation elements. It is based upon the SMIL 1.0 timing model, with some extensions, and will be a true subset of SMIL 2.0.
SMIL 2.0 Specification:
http://www.w3.org/TR/smil20/
SMIL Animation:
http://www.w3.org/TR/smil-animation

Web Accessibility

A revised working draft for W3C's *User Agent Accessibility Guidelines 1.0* was published in June. This document provides guidelines for designing "user agents" that lower barriers to Web accessibility for people with

(continued over)

September 2001
Vol. 7, No. 3

CONTENTS

PUBLISHED QUARTERLY

ISSN 1463-662X

September 2001
Vol. 7, No. 3

interChange is published quarterly by the International SGML/XML Users' Group. *interChange* contains articles and news about the development and use of SGML, XML, and related standards. The International SGML/XML Users' Group is recognized by the International Standards Organization (ISO) as participating in the development of ISO 8879.

The International SGML/XML Users' Group is a registered charity established to promote the use of ISO 8879 and to provide a forum for the exchange of information about SGML, XML, and related standards. The Annual General Meeting is held each year at the XML Europe conference.

Subscription to *interChange* is a benefit of membership of the International SGML/XML Users' Group or its chapter organizations. Membership is open to any interested individual or organization.

Address membership inquiries, advertising orders, change of address notifications, and business correspondence to:

International SGML/XML Users' Group,
Copse House
15 Upton Close,
Swindon, Wiltshire,
SN25 4UL, United Kingdom.
t: 44 1793 721106
f: 44 1793 721106
e: info@isgmlug.org

Send correspondence regarding articles in the newsletter, press releases, and editorial inquiries to Eamonn Neylon by electronic mail at editor@isgmlug.org

President	James Mason
Treasurer	Richard Light
Secretary/Administrator	Yvonne Vine
Editor	Eamonn Neylon
Copyeditor	Bettie McDavid Mason
Webmaster	Robin Cover
Production	Daniel Murphy

ISSN 1463 - 662X

http://www.isgmlug.org/

disabilities (visual, hearing, physical, and cognitive). User agents include HTML browsers and other types of software that retrieve and render Web content. The document will also benefit developers of assistive technologies because it explains what types of information and control an assistive technology may expect from a conforming user agent. The associated *Techniques for User Agent Accessibility Guidelines 1.0* document, which provides a more compact form of the guidelines together with information on how they can be implemented, has also been updated.
Guidelines: http://www.w3.org/TR/UAAG10/
Techniques: http://www.w3.org/TR/2001/WD-UAAG10-TECHS-20010622/

Scalable Vector Graphics Providing the Web with a Better Image

The *Scalable Vector Graphics (SVG) 1.0 Specification* became a W3C Proposed Recommendation in July. SVG is a language for describing two-dimensional graphics in XML that enables the integration of three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), images, and text. Graphical objects can be grouped, styled, transformed, and composited with previously rendered objects. SVG includes facilities for defining nested transformations, clipping paths, alpha masks, filter effects, and template objects. SVG drawings can be interactive and dynamic. Animations can be defined and triggered either declaratively (i.e., by embedding SVG animation elements in SVG content) or via scripting. A free SVG viewer is available from Adobe.
Specification: http://www.w3.org/TR/SVG/
Viewer: http://www.adobe.com/svg/viewer/install/main.html

More XHTML Events

A revised working draft for *XHTML Events: An Updated Events Syntax for XHTML and Friends* was published in June. This specification allows XHTML processors to uniformly integrate event listeners and associated event handlers with Document Object Model (DOM) Level 2 event interfaces. A subset called basic events has been defined for use on simpler devices.
Draft Specification: http://www.w3.org/TR/xhtml-events/

Microsoft and IBM Supplying Schema-Aware Parsers

Version 4.0 of the Microsoft MSXML parser, which was made available for public beta testing in July, will support the full XML Schema definition, providing XSD validation with both SAX and DOM interfaces and a Schema Object Model (SOM) that enables schema information to be accessed through the interfaces. IBM has also updated its XML Parser for Java (XML4J) to support the final version of XML Schemas, SAX 2.0 and DOM Level 2. For more details visit http://msdn.microsoft.com/xml or http://alphaworks.ibm.com/tech/xml4j

---

President's Report: News from Montréal

Extreme Markup Languages 2001

What do we mean by markup languages? In spite of some three decades of using them and a decade and a half since ISO 8879 was adopted, we're still not sure. But we're doing some very creative thinking with SGML and XML nonetheless. This year's Extreme Markup Languages conference in Montréal continued several themes from last year's, particularly the nature of markup languages, schema languages, and the relationship between RDF and Topic Maps.

Last year's conference, with Michael Sperberg-McQueen's keynote on the "Meaning and Interpretation of Markup" and Allen Renear's "The Descriptive/Procedural Distinction Is Flawed," seems to have started a trend of considering what SGML and XML actually mean. This year Wendell Piez took up the theme with his "Beyond the Descriptive vs. Procedural Distinction," taking a rhetorical approach to both the intent and the meaning of markup and providing a new look at validation strategies. In "XML, Stylesheets and the Rematheticalization of Formal Content," Andrea Asperti and colleagues at the University of Bologna examined formal proofs and the means of linking the logical content of mathematical expressions to their presentation. Schema languages for XML have proliferated recently, and at the conference they underwent formal analysis from Henry Thompson and Richard Tobin, as well as from Makoto Murata. (Murata also presented the most recent status of RELAX NG, which merges his RELAX schema language with James Clark's TREX.) The theoretical theme of the conference filtered its way down into specific technologies like Topic Maps, as could be seen in "Towards a General Theory of Scope," by Steve Pepper and Geir Ove Grønmo.

Elaine Svenonius's keynote, "The Intellectual Foundations of Knowledge Representations," which presented an ontology of cataloging and knowledge-representation systems, set the tone for much of the conference. More than a third of the presentations at the conference dealt with knowledge representation, usually through some aspects of Topic Maps or RDF. Indeed, a third of those knowledge-related presentations dealt with Topic Maps *and* RDF. The SGML community started out just trying to capture documents in an enduring electronic format. Now we've moved far beyond that: we're concerned with what those documents mean—and what the documents we can't capture mean. SGML and XML have moved beyond being metagrammars for tagging text and have spawned metalanguages for navigating information.

Another major theme of the conference was information transformation. There were a number of presentations and tutorials on XSL and related topics. Papers like "XSL and Hyperdocuments: Applying XSL to Arbitrary Groves and Hyperdocuments," by Eliot Kimber and his colleagues at DataChannel, show how transformation has moved far beyond simply applying stylesheets to prepare documents for printing.

GCA, the longtime sponsor of this and other related conferences, has become a separate organization from Printing Industries of America. GCA has also changed its name to IDEAlliance.

ISO/IEC JTC1/SC34/WG3

WG3, the SC34 group responsible for Topic Maps, met on Saturday, 11 August. SC34 has several proposals out for ballot on projects related to Topic Maps, so most of the meeting was devoted to planning rather than actual project work. Future development of Topic Maps demands establishing a strong model for how maps work and how they are to be processed. WG3 already has several projects approved or out for ballot that call for the creation of models. The Montréal meeting decided that the model should begin with a core, to which will be added an infoset. Extending the models to show their relationship to UML (Unified Modeling Language) or MOF (Meta Object Facility), which are part of the OMG (Object Management Group) efforts at modeling information structures, is being considered.

The recommendations of the WG3 meeting, with links to other documents, can be found at http://www.y12.doe.gov/sgml/sc34/document/0240.htm.

TopicMaps.org

Shortly after ISO/IEC 13250, the basic Topic Map standard, was released in early 2000, a group of Topic Map enthusiasts set out to develop a specification for an XML interchange representation of the standard. Last winter they released XTM 1.0 and some supporting materials. Earlier this summer the XTM DTD was added to the ISO standard, and much of the work on Topic Map models went back to SC34/WG3. The group decided to become a member section of OASIS and transfer its remaining projects to that venue. The organizational meetings for the new structure were held in Montréal. Eric Freese, the last chairman of the old organization, was selected as the interim chairman of the new organization. The group hopes to have the new structure approved and operating by the time of the XML 2001 conference in Orlando.

An Introduction to RDDL

Jonathan Borden
jborden@mediaone.net
Tufts University School of Medicine, Department of Neurosurgery
New England Medical Center #178
750 Washington Street, Boston, MA 02111

The Resource Directory Description Language (RDDL) was created to answer the question, "What ought a namespace URI reference?" RDDL uses an XHTML format in which resource elements are embedded. Each resource element contains a simple XLink referencing the related resource.

Background

RDDL was developed on the XML-DEV mailing list in response to recurrent debates surrounding the W3C XML Namespaces Recommendation. Ever since the release of this recommendation, there has been controversy about whether a namespace URI ought be a dereferenceable URL and if so, what ought be returned. Though the namespace recommendation does not require that the URI be dereferenceable, URIs were chosen as the mechanism of generating unique names, and common usage of namespace URIs prefixed with "http:" suggests that such URIs indeed are dereferenceable. No guidance had been given as to what, if anything, ought be returned.

In response to yet another periodic heated discussion regarding the role of namespace URIs, in late December 2000, Tim Bray suggested that a namespace URI should dereference to some XHTML with a bunch of XLinks. Within a couple of days, a group of developers quickly settled on the RDDL proposal, and within a single week code to parse the RDDL format appeared. RDDL can now be found at the end of a number of schema-related namespace URIs, including XML Schema and Schematron, document formats

such as RSS 1.0, and software projects such as the XSLT Community Extension Library.

XML Namespaces

A namespace is properly considered a collection of names. Namespaces partitioned by a unique identifier are properly the collection of names, not the partition. The XML Namespace recommendation defines a partition mechanism without defining the mechanism by which names are collected.

The use of URIs as a mechanism of namespace partitioning has been controversial and confusing for its potential user comunities. The computer science community finds that XML namespaces differ from traditional namespaces, which conventionally refer to a set, whereas XML namespaces more properly refer to an unbounded partition. For the Web community the use of URIs which can be resolved (i.e., URLs) is controversial because it suggests that XML namespace identifiers "point to something," yet no guidance is provided on what they ought to reference. RDDL is simple, solves both of these problems, and works with the current Web infrastructure.

The Resource Directory Description Language is an extension of XHTML Basic 1.0 with an added element named *resource*. This element serves as an XLink to the referenced resource, and contains a human-readable description of the resource and machine-readable links that describe the purpose of the link and the nature of the resource being linked to. The nature of the resource being linked to is indicated by the `xlink:role` attribute and the purpose of the link by the `xlink:arcrole` attribute.

The rddl:resource element is defined as:

```
<!ELEMENT rddl:resource (#PCDATA |
 %Flow.mix;)*>
```

That is, it can contain a mixture of text and any of the XHTML elements in `Flow.mix`. The attributes of the resource element are:

```
<!ATTLIST rddl:resource
  id             ID        #IMPLIED
  xml:lang       NMTOKEN   #IMPLIED
  xml:base       CDATA     #IMPLIED
  xmlns:rddl     CDATA     #FIXED
    "http://www.rddl.org/"
  xlink:type     (simple)  #FIXED
    "simple"
  xlink:arcrole  CDATA     #IMPLIED
  xlink:role     CDATA
    "http://www.rddl.org/#resource"
  xlink:href     CDATA     #IMPLIED
  xlink:title    CDATA     #IMPLIED
  xlink:embed    CDATA     #FIXED   "none"
  xlink:actuate  CDATA     #FIXED   "none"
  >
```

A RDDL document is thus an XHTML document into which resource elements have been inserted. The resource elements can contain text or other XHTML elements.

A proper namespace, as understood by the computer science community, is created from an XML namespace as the set of fragment-identifier–qualified URI references contained in the RDDL document describing the XML namespace.

When RDDL is used, a namespace name, as understood by the Web community, references a document that is readable by humans in popular browsers as well as providing machine-readable links to resources related to the namespace. For example, an "XML browser" would be able to locate code or plug-ins needed for the display of namespace-qualified elements contained in embedded SVG, MathML, or other formats.

Describing a Pizza in RDDL

The following is a RDDL document containing a simple resource: an XML Schema for a ficticious "Pizza Description Language" as described by Leigh Dodds (http://www.xml.com/pub/a/2001/02/28/rddl.html).

```
<!DOCTYPE html PUBLIC "-//XML-DEV//DTD
 XHTML RDDL 1.0//EN"
  "http://www.rddl.org/rddl-xhtml.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:rddl="http://www.rddl.org/"
  xml:lang="en"
  xml:base="http://www.bath.ac.uk/~ccslrd/
     examples/pizzaml/ns/">
<head>
  <title>RDDL Document for the Pizza
     Description Language</title>
  <link href="http://www.rddl.org/xrd.css"
     type="text/css" rel="stylesheet"/>
</head>
<body>
  <h1>RDDL Document for the Pizza
     Description Language</h1>
  <p>
  This document provides a list of
  resources associated with the Pizza
  description language, PizzaML.
  </p>
  <rddl:resource
```

```
    xlink:href="http://www.bath.ac.uk/~ccslrd/
       examples/pizzaml/schema.xsd"
    xlink:title="The PizzaML XML Schema">
       <p>
       The XML Schema for PizzaML documents is
       <ahref="http://www.bath.ac.uk/~ccslrd/
       examples/pizzaml/schema.xsd">
    available from here</a>.
       </p>
  </rddl:resource>
 </body>
 </html>
```

In this example, the resource element simply serves as a container for text that describes the material at the other end of the link. A resource element, however, can do more than carry descriptive text that humans can read. It can also indicate to a RDDL processor what roles resource components are to play. For this purpose, RDDL defines *nature* and *purpose* components.

Natures and Purposes

The *nature* of a related resource has to do with characteristics of the resource itself. The *nature* is related to a resource type. (The term *type* has different meanings in different contexts and a different term *nature* was chosen so as not to further confuse an already overloaded term.) A resource *nature* is a URI reference, such as the namespace URI of the root element of the document, or a canonical URI describing the MIME media type of the document. A Nature URI can be any URI and is asserted by the RDDL document describing the resource as the value of the `xlink:role` attribute.

The *purpose* of a related resource is a URI reference that describes the purpose of the resource with respect to the URI being described by the RDDL document. The *purpose* is asserted by the value of the `xlink:arcrole` attribute for the `rddl:resource`.

Extending the Pizza Resource

The Pizza Description Language is augmented by the addition of several resources which describe the XML Schema as well as several XSLT transforms converting from the Pizza Description Language into XHTML and into RSS 1.0:

```
<rddl:resource
    xlink:role="http://www.w3.org/1999/XSL/
       Transform"
    xlink:arcrole="http://www.w3.org/1999/
       xhtml"

    xlink:href="http://www.bath.ac.uk/
       ~ccslrd/examples/pizzaml/
       pizza2html.xslt"
    xlink:title="Transform Pizza Menu to
       XHTML">
<!-- ... -->
</rddl:resource>

<rddl:resource
    xlink:role="http://www.w3.org/1999/XSL/
       Transform"
    xlink:arcrole="http://purl.org/rss/1.0/"
    xlink:href="http://www.bath.ac.uk/
       ~ccslrd/examples/pizzaml/
       pizza2rss.xslt"
    xlink:title="Transform Pizza Menu to
       RSS">
<!-- ... -->
</rddl:resource>
```

More resources can be defined to establish contexts for validation:

```
<rddl:resource
    xlink:role="http://www.isi.edu/in-
       notes/iana/assignments/media-types
       /text/xml-dtd"
    xlink:arcrole="http://www.rddl.org/
       purposes#validation"
    xlink:href="http://www.bath.ac.uk/
       ~ccslrd/examples/pizzaml/pizzaml.dtd"
    xlink:title="The PizzaML DTD">
<!-- ... -->
</rddl:resource>

<rddl:resource
    xlink:role="http://www.w3.org/2000/10/
       XMLSchema"
    xlink:arcrole="http://www.rddl.org/
       purposes#schema-validation"
    xlink:href="http://www.bath.ac.uk/
       ~ccslrd/examples/pizzaml/schema.xsd"
    xlink:title="The PizzaML XML Schema">
<!-- ... -->
</rddl:resource>
```

Well-Known Natures and Purposes

RDDL defines a hierarchy of generally useful *natures* and *purposes* at the URLs http://www.rddl.org/natures and http://www.rddl.org/purposes. These documents themselves serve as examples of using RDDL for defining RDDL terminologies.

The URI prefix http://www.isi.edu/in-notes/iana/assignments/media-types/ serves as a root URI for media types and allows creation of a nature URI for any media type. For example, http://www.isi.edu/in-notes/iana/assignments/media-types/application/xml-dtd serves as an established URI describing the nature of a DTD.

Resource elements that are intended to indicate the *nature* of HTML can point to http://www.isi.edu/in-notes/iana/assignments/media-types/text/html for generic HTML, to http://www.w3.org/TR/html4/ for HTML 4.0, to http://www.w3.org/TR/html4/strict for HTML 4 Strict, and to http://www.w3.org/1999/xhtml for XHTML.

Using RDDL in Distributed Web Applications

XSLT

An XSLT stylesheet may transform the content of a document from one *nature* to another. The *nature* of an XSLT stylesheet is indicated by http://www.w3.org/1999/XSL/Transform. The purpose of an XSLT stylesheet is the *nature* of the result of transforming an input document having a *nature* of the base URI of the XSLT resource, typically the base URI of the RDDL document, by the stylesheet.

This convention provides for the possibility of generating a transform of A→C given known transforms from A→B and B→C. Such a transform might be automatically generated by chaining through the RDDL documents describing the namespaces for A, B and C.

SAX Filters

Similarly, an XML processing pipeline might be generated by looking for RDDL resources within namespace URIs labeled as SAX filters implemented in whatever language desired. For example, Java and Python implementations of a particular namespace-related behavior can be implemented and referenced side by side within a RDDL document at the namespace URI.

RDDLClassLoader

The feature of referencing code within a namespace URI is made use of by the RDDLClassLoader Java class. This class implements a Java class loader that "finds" Java classes having a particular purpose within a namespace URI. This class is almost trivially simple:

```
/*
 * RDDLClassLoader.java
 *
 * Created on March 3, 2001, 10:55 AM
 */

package org.rddl.helpers;
import java.util.SortedMap;
import java.util.Vector;
import java.util.Iterator;
import java.net.URL;

import org.rddl.Resource;
/**
 *
 * @author  Jonathan Borden
    <jonathan@openhealth.org>
 * @version
 */
public class RDDLClassLoader extends
 java.net.URLClassLoader {
    static final String STR_NATURE_JAVA =
       "http://www.rddl.org/natures#java";
    static final String STR_NATURE_JAR =
       "http://www.rddl.org/natures#JAR";
    /** Creates new RDDLClassLoader
     * @param nsUrl the namespace URI
     * @param purposeURI The
 <code>purpose</code> of this ClassLoader
 connection
     * @throws IOException
     * @throws SAXException
 */
    public RDDLClassLoader(String
 nsUrl,String purposeURI) throws
 java.io.IOException,
 org.xml.sax.SAXException{
       super(buildUriList(nsUrl,
       purposeURI));
    }

/** This is an internal static method that
 creates a URL array from the RDDL URI and
 purpose
 * The <code>nature</code> is either "java"
    or "JAR"
 * @param URI The namespace URI
 * @param purposeURI The RDDL <code>pur
    pose</code>
 * @throws IOException
 * @throws SAXException
 * @return URL[] - an array of URLs -- this
    is typically passed to the
 * constructor of URLClassLoader()
 */
    protected static URL[]
 buildUriList(java.lang.String
 URI,java.lang.String purposeURI) throws
 java.io.IOException,
```

```
org.xml.sax.SAXException {
       org.rddl.Namespace ns =
       RDDLURL.getNamespace(URI);
       java.util.SortedMap ress0 =
ns.getResourcesFromNature(STR_NATURE_JAVA);
       java.util.TreeMap ress = new
       java.util.TreeMap(ress0);
       ress.putAll(ns.getResourcesFromNature
       (STR_NATURE_JAR));
       java.util.Vector strArr = new
       java.util.Vector();
       Iterator iter =
       ress.values().iterator();
       while(iter.hasNext())
       {
           Resource res =
           (Resource)iter.next();
           if (purposeURI.equals
           (res.getPurpose()))
                  strArr.addElement
                  (res.getHref());
       }
       int len = strArr.size();
       URL[] uris = new URL[len];
       URL baseURL = new URL(URI);
       for(int i=0;i<len;i++){
           uris[i] = new URL(baseURL,
           (String)strArr.elementAt(i));
       };
       return uris;
   }
}
```

Summary

RDDL provides a description, readable by both humans and machines, of a namespace referenced at the namespace URI. As the Semantic Web gets created, we feel it is important for this new Web to be built in a fashion compatible with the current HTML-based Web. By design, RDDL based on XHTML is readable in common browsers (which conveniently ignore the rddl:resource elements); and, because it contains XLinks, it is potentially readable by software programs.

The XML community has desired an XML browser that can display and manipulate any sort of XML document. RDDL enables creating such a browser by allowing dynamic download of new behaviors attached to namespaces. Such changes in behavior would, for example, enable a browser to accept MathML or SVG elements in XHTML.

Despite misgivings among its intended audience, the XML Namespaces recommendation has achieved widespread adoption in the marketplace. RDDL's creators hope that it is an acceptable solution to some of the problems the community has had with namespaces. By defining a reasonable solution to what a namespace URI might reference, RDDL allows XML namespaces to be used in more powerful ways than as mere unique identifiers for disambiguating elements names across vocabularies.

Dr. Jonathan Borden is an editor of the RDDL specification. He is the director of the Open Healthcare Group and a co-chair of the ASTM E31.25 XML Healthcare DTD subcommittee.

XML Configuration Management

Jim Gabriel
jim@barbadosoft.com
Barbadosoft BV
Stephensonstraat 19, 1097 BA Amsterdam, The Netherlands

Complex XML-based systems are easy to build but inherently difficult to maintain. Keeping complex applications alive can sometimes become so difficult that stagnation is preferable to change. Taking a leaf out of the database gurus' book can shed a different light on XML. The complexity is not the problem. Managing the complexity intelligently is the problem. A solution from the 1970s, perhaps?

Modeling: A Problem in XML

Carla Corken of DataChannel opened a presentation at XML Europe 2001 in Berlin with an interesting but highly challenging statement: "The problem with XML is that you can't model it." Corken's

presentation went on to discuss a range of issues in XML application development projects, focusing in particular on how complexity in an XML-based application environment can rapidly spiral out of control in a way that is out of all proportion to the requirements.

That complexity can be so painful will come as no surprise to any owner of a sophisticated XML-based application.

Imagine any environment in which multiple schemas are used (or one schema evolves through multiple versions), and in which most or all of the processing is automated—the transformations, the publishing, the business processes acting on the data, the user profiling, and so on. For example, take the XML-driven online equivalent of a traditional printed newspaper. Or a business-to-business marketplace. Or the customer service portal for a major service provider such as an ISP or telecommunication company.

Now, measure the cost of building that environment from the perspective of the person who cares about the XML. This is no simple task. How many DTDs or equivalent schemas need to be created? How many scripts, Java classes, executables, workflows, and source control systems will you need to craft? What will you use to control the structure of the packets of XML that flow around the system so that they can be manipulated efficiently at development time and rapidly integrated with other processes? How do you ensure a consistent and yet flexible approach to style, localization, versions, evolution, and knowledge management? In short, how do you model the XML part of the application? The simple answer is that you don't. You can't. Only experience can tell you what is needed.

One way to gain an understanding of the enormity of the total cost and the extent of the configuration management problem is to measure what it would take to implement a relatively simple change to an object used in the environment once it is up and running, and then extrapolate that to the sum of all the objects in the whole system. For example, take an element that is commonly used and convert it into an attribute of the parent element. Give the attribute four possible values, one of which is the default. This sounds like a simple request, but it can be devastating in an complex environment that has already gone into production.

Chart all the places where you need to go and patch things up, including any live data created according to the old definition. How would you make the change and go live again as efficiently as possible? Would you create a test environment where you could modify the relevant schemas and observe what broke? Usually, trial-and-error figures high on the list of techniques at this point. Ask yourself if this is a reliable way of working. Would you feel comfortable delegating the task to somebody less experienced? Could anybody in your team undertake the work? Could you hire a system integrator from another company to do the work? Ask yourself if there is any software available to help you conceptualize the entire environment in the first place and allow you to model the change. Ask yourself what happens to an organization when the person who understands the conceptual knowledge of a system gets up and leaves.

The premise for a solution is simple: If you can model something, you can usually automate much of the process of building or implementing it. If this is true, you should be able to change anything in the model and rebuild the thing that your system previously built automatically. But if, as Corken states, you cannot model XML, then this is futile conjecture. Many will point to UML as the answer. Many hardened software developers, however, object to the obscure and overly technical DTDs that UML tools produce and point out, quite correctly, that UML-based tools are always only the starting point. Making a round-trip through the model to rebuild applications is no longer possible after the first build because typically so much is changed that the UML model is hopelessly out of sync. (And the jury is still out on XMI.)

An Historical Parallel: Developing the Three-Schema Model for Databases

Let's examine how modeling entered the world of a completely different set of IT problems, the relational database. Way back in the mid-1970s, the world was embracing relational database technology as rapidly as it is now embracing XML. Certain parallels exist between these spurts of progress in information technology. For example, relational databases could not initially be "modeled" using software to build tables and relationships and to apply the constraints necessary to enforce a set of business rules. Computer Aided Software Engineering (CASE) was the popular answer to that particular problem, although round-tripping through the model in the CASE tool to implement changes invariably didn't help when changes were needed.

The fundamental problem is this: the relational

data model gave us a way of structuring data. There is no mechanism in the relational model for capturing business processes and enforcing business logic in a relational database. Events are not described in the relational model, unless you consider the Delete event and the possible referential integrity constraints for related data (Cascade, Nullify, Restricted). The more expensive application development systems (or very highly paid programmer/consultants) generally provided the missing parts of the puzzle, and often those parts were not to be found in the database itself. Business logic is taken care of in C programs, Perl scripts, or proprietary 4GL applications, for example, none of which can be charted by a modeling tool in order to track the interdependencies and autogenerate entire systems. Modeling a relational database is therefore meaningless. On the other hand, generating a relational database from some other form of model—the output of a CASE tool, for example—is sensible.

A database, just like an XML document, is described by a schema. A schema describes structure. If you change the schema for a database, you are effectively changing the description of the tables and relationships in that database. And as any database designer will testify, changing the schema usually trashes the data in the database. To fix the situation, you export the affected data, drop the tables, massage the data offline using anything you can think of (from Perl to an Excel spreadsheet), recreate the tables according to the revised schema, and reimport the data. If everything has gone according to plan, you have a healthy database again. However, a database is nothing without an application to use it. Your change in the database will break the applications using the database. You will typically have a lot of work ahead of you before the entire universe of that database is synchronized with your new version of reality.

One of the best initiatives to emerge from this conundrum of shortcomings in the 1970s was the three-schema architecture. Published in 1975 by the ANSI X3/SPARC study group, a seemingly innocuous document entitled *Study Group on Database Management Systems: Interim Report 15-02-08* went on to be adopted by ISO and find its way into almost every undergraduate computer science degree course anywhere in the world. The proposal made by this document was that database engineers should adopt a "three-schema architecture" to make their systems easier to build, deploy, and manage.

The three schemas of database applications are the conceptual, the external, and the internal. The conceptual schema describes the structure of the database for one or more applications. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. The internal schema describes the physical storage in the database. The external schemas (there can be several) are equivalent to logical users' views—that is, forms, windows, printed reports, services, and so on. Each external schema describes the part of the database that a particular user group (or software application) is interested in and hides the rest of the database from that user group.

The single most important principle of the three-schema architecture is that none of the end-users of an application needs to understand either the conceptual or the internal schemas, and none of the designers and developers of the application needs to care about the internal schema (although most database administrators would disagree!). In order to facilitate this blissful ignorance of the participating schemas, a powerful set of interfaces needs to be created to make autogeneration of schemas possible. Building a database application means building a conceptual schema, then using those definitions to build external schemas (much of which could be autogenerated), and generating the relevant database in whichever database management system (or systems) is being used.

In other words, a conceptual model of a particular business requirement should result in a set of object descriptions that can be used to generate the relevant tables and relationships in *any* proprietary database management system and also provide the core set of definitions in any user interface, regardless of the hardware or software used to host the application. At run time, three-schema architecture assumes the presence of interface or driver mechanisms for any third-party hardware or software, which means that any network protocol should be feasible, any operating system, or any computer. A runtime engine, or virtual machine, must exist to handle the flow of data between various schemas and control the reaction to events in the application.

In essence, the ANSI study document formally describes the way in which database applications should work: somebody designs an application, a database administrator creates tables and relationships to support that design, a user interface designer creates a set of forms and windows that allow the users to interact with the application, and data is stored in the database. No reinvention of the wheel occurs at any stage, and every second of every

minute invested in creating a conceptual definition is capitalized upon as efficiently as possible according to good "write once, read many"(WORM) principles.

For example, by capturing the business requirements for handling sales tax at a conceptual level in the conceptual schema, any form that relies on that logic to correctly handle user input can be built from those conceptual definitions without the form designer having to write the logic out all over again in procedural code in every form. Similarly, the database requirements for handling sales tax correctly can be met automatically in any database system, from creating the tables and relationships in the first place to a run time situation of indexing relevant columns and enforcing referential integrity constraints when the user deletes records.

The heart of three-schema architecture is the ability to autogenerate from models and to provide an interface-driven infrastructure for highly open and highly maintainable applications. A three-schema compliant application can switch database back ends and GUI front ends at will. A major change in a three-schema compliant application (such as moving the rate of sales tax from 17.5% to 19%) should be possible by changing one single definition in one place in the conceptual schema, regardless of where it is actually used in other schemas, and then autogenerating everything else. This is true "magic button" technology.

Nowadays, in the era of J2EE technology and application servers, this open systems architecture is nothing special. Back in 1975, however, before client-server had even been dreamed of, it was revolutionary—so revolutionary, in fact, that it is perhaps no small wonder that the study document did not recommend a product architecture to back up that description. Commercially, three-schema architecture entered not with a bang but a whimper. The world should have reeled, but in reality the only beneficiaries were students and, much later, the aficionados of n-tier application development principles.

To my knowledge, the only commercially successful adoption of three-schema architecture was by Uniface, the Dutch open 4GL application development system launched in 1987 (and now owned and sold by Compuware Corporation). While sometimes tricky to design and build, Uniface applications are simple to maintain, and can be deployed against virtually any database management system, across any network architecture, on any operating system, and with any graphical user interface. In the case of Uniface, the principles of three-schema architecture were used to make an open, maintainable system an affordable reality.

Towards a Model for XML Systems

Let's fast-forward to the present day. The problems of configuration management in complex XML-based applications are horrendous. A single object at the design stage (such as a transaction ID) is used or referred to in so many different places by the time the application has been built that the application owners can easily spend all of their maintenance and new development budgets in trying to fix the unexpected bugs that are the results of a minor change.Our inability to predict the effects of a change, as Corken will agree, has a lot to do with our inability to model XML.

Many will argue, however, that the only constant in the history of information technologies has been the need for change. As XML comes of age, it faces the same set of requirements and problems that apply to every other technology developed for organizational systems. A key problem is tracking the versions in the evolution of systems as they adapt to embrace new business needs. This problem alone, it would appear, lies in the path of anyone attempting to draw up a powerful object model for XML-based applications. Conceptualizing and automating the process of configuration management requires powerful version control and schema evolution, one of the major headaches even for three-schema architecture (solvable only via repository technology and highly sophisticated source control systems operating on objects at a very fine level of granularity).

So is it ever going to be possible to model XML in such a way that a conceptual schema may be built? An explosive increase in the use and acceptance of XML and its application in areas not anticipated by its original designers have led to a staggering array of interrelated specifications, standards, and incentives by industry consortia to address a growing set of demands on the technology. Things are getting more complicated by the day.

The principles of version control and the headaches of change and schema evolution are well understood as the result of many years of accumulated knowledge and experience with the many new technologies that have arisen since the advent of the "computer age." On the other hand, there are specific problems of change and evolution facing XML technologies that differ from the same set of problems in the past. The extra considerations lie not

only in the nature of XML itself, but also in the ways it is being used and the new application architectures in which it plays a key role.

Most XML application owners will happily agree that autogeneration of entire environments after making a simple change to a definition is not really relevant. The most important thing is the ability to conceptualize the interdependence of all the objects used in a system—from the transaction ID in a DTD to the credit check carried out by a Web service when a new customer tries to purchase more than US$ 50K worth of goods—and to safely and reliably know that anybody can walk onto their project and rapidly gain a conceptual understanding of all its various pieces. It is also important to know how to make changes while preserving the integrity of your data and definitions. Again, this requires a model.

One of the problems facing the academics, however, is that XML is growing so fast: trying to model something that is developing so rapidly is half the challenge. The early explorers have proven and expanded the enormous potential of XML. As with many new ideas (such as relational database theory), adoption of technologies goes through adolescence faster than the development of tools to support it. This happens for many reasons—inevitable changes and oversights result in a rapid evolution of standards offering a moving target against which to work. As these standards have stabilized, early adopters who have built prototype or limited scope applications using XML are now attempting to roll out solutions, expand the scope of projects, or migrate to a new version to take advantage of lessons learned in early phases.

Let's go back to basics for a moment and attempt to chart the parts of XML that a modeling paradigm must address. XML had its origins as a metalanguage for describing the structure of documents in environments where any or all of the following are true:

- A range of publishing formats must be supported from common sources.
- Strict requirements must be met on the content of documents.
- Contributors from many sources must be able to update information.

In common with other storage data formats, such as relational data, or transfer formats, such as CSF or EDI, XML has the ability to define whether fields may be mandatory or optional. With the emergence of more data-oriented schema languages it now also allows the specification of exact cardinality ranges. In addition to these constraints, XML also defines ordering and makes a distinction between attributes and elements as a means of specifying field sets.

The document-centric nature of XML makes the location of particular fields within a data set less absolute than, for example, relational data. Mappings between an intermediate XML format and existing storage formats exacerbate this problem. Managing evolution in systems using XML requires management of the definitions that are encapsulated in the schemas which describe them.

XML is easy to generate and consume in an ad-hoc manner, but XML without rules is impossible to manage in any coherent way. As with any other data format, data structures in a complex system *must* be defined by a schema. XML schemas can be very difficult to write and maintain because any level of modularization requires the creation of a complex web of files with no formal way to describe the dependencies between them. Sophisticated XML applications require strict data structures to be defined in schemas. This makes processing predictable. XML, however, has a lower tolerance for change than other data formats. XML data often becomes unusable when changes to business rules dictate schema modifications.

Ultimately, however, XML has no standard mechanism for versioning, a consequence of the industry's inability to model XML at a canonical level. Simple file comparisons tell us nothing. Current debate in forums such as the technical xml-dev list makes it clear that management of multiple live versions of a system is being handled in an ad-hoc manner.

For example, one of the rapidly growing uses of XML is as a medium for intersystem communication and application integration. This has led to new ways of looking at modeling and deploying applications. Systems may be modeled based on message formats with XML schemas being used to describe the interfaces of distributed components. This raises complex evolution issues. Systems must simultaneously support the rollout of new versions within an organization and backwards or forwards compatibility with existing processes or data sets. Because applications are now being made available to external users and must deal with data over which the organization has no control, new versions must be communicated to those users in a way that allows them to determine when and how they will incorporate these changes, and systems must be able to recognize by some means the set of validation rules that are applicable to an incoming message.

There are a number of initiatives to provide platforms for publishing schemas that represent interfaces to public services and standard vocabularies. The challenges that remain include (1) publishing multiple versions and localizations, (2) providing information about cross-version compatibility, (3) updating common components universally, and (4) offering dynamic access to work in progress.

When making changes to an XML schema in a *production* environment, the implementer must consider a very long list of issues. For example, do the changes result in a stricter or more relaxed schema? How does existing data need to be modified in order to conform to the new schema? What should we do with existing data and application software?

Metadata as a Key to an XML Model

At first glance, the issues appear daunting. To a data modeler, however, they represent a great challenge. The scent of a solution is in the air, and it smells of abstraction, inheritance from multiple object classes, meta-object models, and magic button technology. Fundamentally, the missing piece of the current configuration management puzzle with XML is the conceptual schema. Remember, the conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. It is the conceptual schema in a database application environment that enables high-speed change with minimum resources.

From a conceptual schema, everything else should be a down translation. DTDs, schemas, stylesheets, Java classes—nothing should be so complicated that a conceptual schema cannot act as an engine (or engine driver) to build whatever is needed to make the conceptual definition an external or internal reality. Building XML databases should be an automated step. Modifying databases when a definition is modified in the conceptual schema should also be an automated step, or at least something that can be managed intelligently with software and fed into a good workflow process.

And last but not least, a conceptual schema allows an application owner to conceptualize the entire application environment. This allows a newcomer to a software development team to rapidly discover where everything has been used or referred to and to model the changes in any given configuration before changing anything in a deployed application. The result: impact analysis and automated change management utilizing the most sophisticated schema repository the world has ever seen.

However, the implementation of three-schema architecture assumes that an agreement has been made about a modeling language or protocol for the conceptual schema. In the case of relational databases, the relational model provided the starting point. The relational model, as we have already discussed, falls a long way short of supporting business processes, workflow, and other important aspects of the software that is built around a relational database. By extending the model to allow the missing but highly interdependent other parts of the puzzle to be described in the conceptual schema (e.g., events, procedural code to react to events, and user interface properties), application developers are able to generate not only databases but also the user interfaces, the invisible software for handling the business processes, the configuration files, and so on. In modeling terms, the data model of a database is described at one or more levels of abstraction.

Metadata (that is, data about data), is the key. Metadata makes is possible to describe how each object is used in any place and to chart the relationships between that object and every other relevant object. Capturing the metadata in a relational database (a data dictionary, now more commonly known as a repository or application model) provides application developers with the wealth of information that they need to build a true conceptual schema. A powerful conceptual schema provides model-driven schema evolution, impact analysis, change management, data validation, and data conversion, to name but a few.

This is all well and good for relational database applications. However, is the three-schema architecture good for XML? Certainly, although strictly speaking the documentation of the architecture would need custom-fitting to cater for XML. Is it possible to model XML? Using the principles of three-schema architecture, it would certainly seem so. The trick lies in building a model that answers the needs of application developers and then defining a mechanism for capturing and exploiting it. The end justifies the means.

As for a model for XML, we are only just beginning. The XML Schema recommendation provides us with an object model for XML. (DTDs, on the other hand, offer no modeling capability whatsoever.) XML Schema, however, is the equivalent in modeling terms to the early relational model: documents and their parts (complex types, model groups, et al.) can be described very thoroughly indeed, but all the important parts of the *application* of those

documents are missing. The trick is to take the XML Schema and extend it (considerably) with the metadata that describes those missing parts. Only then can we talk of true model-driven schema evolution, impact analysis, change management, data validation, and data conversion. This is a powerful area of application technology that will probably not be realized by any of the major vendors in the XML software market.

Jim Gabriel (b. 1961) was educated at the University of Kent, Canterbury, and the University of California, Santa Barbara, focusing on English literature. He moved into IT in 1985 via marketing and technical writing. Learning to program in various languages using various tools, he found he had a predilection for database applications with highly graphical front ends. He has written technical documentation, developed training materials, and taught courses in IT subjects. Having burned his fingers badly on configuration and change manage - ment of SGML and XML, he founded Barbadosoft in 2000 to build CorteXML™, an XML manage - ment platform that enables high-speed change in complex XML environments.

Topic Maps and RDF

Eric Freese
eric@isogen.com
Director of Professional Services, Midwest, ISOGEN International/DataChannel
1611 West County Road B, Suite 204, St. Paul, MN 55113
www.isogen.com

Background

Over the past couple of years, very similar claims have been made for the W3C's Resource Description Framework (RDF) and Topic Maps (ISO/IEC 13250:2000 and XML Topic Maps (XTM)). The more zealous supporters have promoted each as the absolute-best way to associate arbitrary metadata with arbitrary content and to support an unbounded variety of information finding and other functionalities. Indeed, both have been openly described by respected pundits as panaceas for every kind of information management woe, but rarely by the same pundits.

At the Extreme Markup Languages 2000 conference, held in Montreal, Canada, last summer, proponents of each methodology spoke about them, with the amazing discovery that perhaps the models had more in common than most people imagined. This article will briefly explain each methodology, then compare the similarities and the differences, and finally provide an update on the progress and events that have occurred since last year's Extreme conference.

Topic Maps: An Introduction

To put it in a nutshell, Topic Maps are made up of topics, occurrences of topics, and associations among topics. Information about the topics can be inferred by examining the associations between topics and occurrences that are linked to the topic. The collection of these topics and associations is what is referred to as a Topic Map.

Topic Maps, as defined in ISO/IEC 13250:2000, and XTM, as defined by TopicMaps.org, are used to organize information in a way that can be optimized for navigation. Topic Maps can be thought of as the online equivalent of printed indexes. Topic Maps are also a powerful way to manage link information, much as glossaries, cross-references, thesauri, and catalogs do in the paper world.

Topic Maps are built of units called topics. In linguistic terms, a topic can be anything that is a noun (for example, the president of the United

States). More generally, a topic is anything a user would like to describe in some way. A topic can have many links that point to all its occurrences (documents and Web pages about the office of the president).

A topic can also have one or more occurrences within a set of information. In terms of the president, occurrences of a topic may be in various items such as speeches, policy statements, or Web sites. Such occurrences are generally outside the Topic Map document itself (although some of them could be inside it), and they are pointed at by using whatever mechanisms the system supports, typically HyTime addressing or XPointers.

Topics can be related together using associations which can express a given semantic (for example, "George W. Bush is the president of the United States"). Topic Map designers can define any kind of semantics for topic associations. Associations are ordinary links that are constrained to relate only topics together. Because they are independent of the source documents in which topic occurrences are to be found, these associations represent a knowledge base that contains the essence of the information, actually representing the essential value of the information. An unlimited number of topics can be associated within topic associations. Each topic in an association is assigned a role that states its interaction with the other topics. For example, George W. Bush may have the role of "holder" and president may have the role of "office" and United States may have the role of "country".

Topics can have various characteristics assigned to them: *names, occurrences,* and *roles*. These characteristics are considered to be valid within certain limits called *scopes*. A topic's scope allows a processing system to avoid ambiguities between similar topics and their characteristics. Any assignment of a characteristic to a topic is considered to be valid within certain limits. For example, in order to distinguish between the president of the United States and the president of a corporation, scopes can be defined based on the themes "politics" and "business."

RDF: An Introduction

RDF provides a model for describing resources. Resources are named using Uniform Resource Identifiers (URIs). As defined in the recommendation, anything can have a URI (including George W. Bush, the United States, and the office of the president). So RDF can, in theory, describe almost anything. However, it is mainly intended to handle resources that can be retrieved over the Internet.

RDF works by making statements about the resources being described. Each statement consists of at least three properties: a subject, an object, and a predicate. Each of these properties is also a resource. Property types identify the properties associated with resources, and property types have corresponding values. Property types express the relationships of values associated with resources. In RDF, values may consist of fixed values (text strings or numbers or whatever) or refer to other resources, which in turn may have their own properties. A collection of these properties that refers to the same resource is called a *description*.

Consider the following statements: "The president of the United States is George W. Bush" and "George W. Bush is the president of the United States." To the human reader, the above statements convey the same meaning (that is, George W. Bush is the U.S. president). To a computer, however, the statements are merely different strings. The human's ability to extract meaning from varied syntax is much different from the capabilities of the computer. RDF uses a three-valued model of resources, property types, and corresponding values to express the semantics.

In order to enable machine processing, RDF expresses semantic information by associating properties with resources. So, before anything about George W. Bush can be said, the data model requires the declaration of a resource representing him. Thus, the data model corresponding to the statement "the president of the United States is George W. Bush" has a single resource (United States), a property type (president) and a corresponding value (George W. Bush).

The RDF data model additionally provides for the description of other descriptions. Often it is important to assess the credibility of a particular description (e.g., "*The New York Times* announced that George W. Bush is the president of the United States"). In this case the description tells us something about the statement "George W. Bush is the president of the United States": that the *Times* asserts this to be true. Similar constructs are also useful for the description of collections of resources. e.g., "The United States government is made up of the legislative, judicial and executive branches." Although the statement is more complex, the same data model is applicable.

A Comparison of Topic Maps and RDF

Topic Maps and RDF are similar in that they both attempt to alleviate the same problem of finding the desired information in a sea of other, mostly useless, information. They both do so by annotating information resources by reference, not within the items being described. There are several other major similarities: (1) both models are very simple and elegant at one level, but thanks to recursion both are also extremely powerful; (2) both models are designed to be extensible; and (3) both models can be used to build semantic networks of information. (A semantic network is a knowledge representation technique that applies a semantic link between two nodes that represent objects or concepts.)

Initial discussion of the two models at the Extreme conference last year led to the surprising discovery that, at the surface, there do not appear to be many significant differences between the two models, other than serialization syntax. The underlying data models for the two seem to be very similar. The use cases also appear to be very similar.

The one main difference is that in RDF Schema, RDF has something Topic Maps don't (yet): a standardized way of expressing an ontology and the constraints upon it. However, a proposal for a Topic Map schema language was offered at the conference.

After some early discussions, it seemed possible to model most of the concepts within Topic Maps using RDF and vice versa. Topics could be regarded as RDF resources, and associations between two topics could be used to build an RDF triple. Occurrences could be modeled by another RDF triple, in which either the resource or the value of its property would be a real information resource as opposed to an abstract topic.

Recent Developments

Much of the hallway discussion after the RDF/Topic Map presentation in Montreal last summer had a sense of excitement that the two methodologies may, at a minimum, interchange—or possibly even unite. As work on the Semantic Web has started, representatives from the RDF community and the Topic Map community have had ongoing discussions about the two models to chart the path forward for each individually as well as to ascertain how the two could intermingle. These discussions have compared the models and determined where the strengths and weaknesses of each exist. A goal is to provide a road map for integration and interchange of the information both can represent. Several implementers have demonstrated how Topic Maps and RDF can both be used to model the same information.

Earlier this year, the XML-based representation for Topic Maps known as XTM was published. XTM has been considered a possible candidate as the unified next generation for both models. There is also discussion about extending RDF even further to support the goals of the Semantic Web. Work continues on building a community around the XTM specification, and several vendors have products that support that specification. Additional standards work is under way to define a query language for Topic Maps and also a constraint language for defining the semantics and intended uses of Topic Map constructs within a given Topic Map.

A conference entitled "Knowledge Technologies," held in Austin, Texas, in March 2001, brought together people from several areas of information management in a way that had rarely happened in the past. This gathering allowed attendees to bridge the differences between models and methodologies to determine where common work was being done. It will be interesting to see how much is accomplished by the second conference, planned for next spring in Seattle, Washington.

Helpful Links

RDF: http://www.w3.org/RDF/
XTM/Topic Maps: http://www.TopicMaps.org
Semantic Web: http://www.w3.org/2001/sw

Eric Freese, chairman of TopicMaps.org, is a senior consultant for ISOGEN International, a DataChannel company. Previously, he was the president and founder of the Electronic Data Foundry, which was recently acquired by DataChannel. He has over twelve years of experience in the area of information and document management, with emphasis on SGML, XML, and related standards. He has developed and implemented training programs and materials from elementary to graduate level. He also has research experience in human interface design, graphics interface development, and artificial intelligence.

Acrobat and Structured Documents

David Penfold
penfold@eps-edge.demon.co.uk
Edgerton Publishing Services
30 Edgerton Road, Huddersfield HD 3AD, United Kingdom
www.eps-edge.demon.co.uk

Earlier this year, Adobe released Acrobat version 5.0, based on version 1.4 of PDF (portable document fomat). When the first version of Acrobat was released, back in the mid-1990s, one of the questions that was asked was, "When is Acrobat going to incorporate document structure and become SGML compatible?" The answer at the time was something like "eventually," and few held out much hope that this stage would ever be reached.

Well (and I was one who doubted), we were wrong. PDF version 1.4 and Acrobat 5.0, which uses PDF 1.4, have finally moved away from the original PostScript foundation and incorporate structure. This is probably not entirely unrelated to Adobe's purchase of FrameMaker and, with the release of Acrobat 5.0, Adobe at last seems to have decided where FrameMaker fits into its product portfolio. However, it is not the purpose of this article to discuss Adobe's marketing plans, rather to discuss how Acrobat and PDF have begun to move into the world of XML and structured documents. For more detail see *Portable Document Format: Changes from Version1.3 to1.4* (Technical Note #5409; http://partners.adobe.com/asn/developer/acrosdk/docs/filefmtspecs/PDF14Deltas.pdf)

There are three main aspects of the changes to PDF. The first is related to forms; the second to what is now described as tagged PDF; and the third the use by Adobe of XML to define metadata, which is also an important aspect of tagged PDF. I will discuss the three in turn.

Forms

Forms were introduced with Acrobat 4 as part of PDF version 1.3. In the new version, however, the functionality is greatly increased. Forms can be linked to a database or to a Web server, so that forms can be completed over the Web. We are all, of course, familiar with HTML forms, but Acrobat forms have the big advantage that the format remains as the format designer intended and, certainly more important in some environments, they can carry digital signatures (increased security is a big feature of Acrobat 5.0, although not an aspect I shall discuss here). It is perhaps a drawback that Acrobat has to be running within a browser in order to submit data and therefore that the Acrobat plug-in has to open within the Web browser before the form can be displayed. One cannot submit data from within the Acrobat Reader (still free), which is perhaps a pity, since submission of forms from Acrobat Reader would have meant that forms could have been available directly on a network or perhaps by e-mail.

The new functionality makes it easier to base forms on existing printed forms by using scanning. (Incidentally, there is now a plug-in that can be downloaded from the Adobe Web site that allows a reasonable number of scans a day without requiring the additional purchase of Acrobat Capture, now no longer bundled with Acrobat; however, major scanning projects will still require Capture.) The ability to include Javascript means that dynamic forms can be created with new fields automatically filled in on the basis of input from other fields and reference to a database or even a spreadsheet file. Other new features are data validation, automatic calculation (addition, subtraction, etc.), and spell-checking within form fields.

As Adobe itself noted, there was no support for XML in the previous release. Data was exported in a proprietary format called FDF (forms data format), which was not very friendly. To be fair, Adobe provided (and still provide) a lot of information for developers on its partners site (http://partners.adobe.com) about how to utilize data in this format, but it could hardly be regarded as an open format. Now, while it is still possible to export in FDF, one can also export in XFDF (XML-based form data format), which is not only much simpler and even readable, as one would expect, but will mean that integration of Acrobat form data with

other applications will be more straightforward. In order to export as XML, one needs to download the SaveAsXML plug-in from the Adobe Web site. The plug-in also allows the document to be saved as HTML. For detail on SaveAsXML, see http://download.adobe.com/pub/adobe/acrobat/misc/5.x/DeveloperInfo.pdf. The XML seems to be based on the names given to data fields, so it is effectively within the programmer's control.

Obviously, Adobe envisages the use of forms in what are conventional contexts, such as customer inquiries, system maintenance, etc. However, Acrobat 5.0 may provide a data input path that is easy to set up for XML-based applications. After all, much of the underlying work has already been done by Adobe.

Tagged PDF

Tagged PDF has been introduced to solve a number of problems, which Adobe brings together under the term "making accessible." The most important of these are: making documents available to people with visual handicaps (the document is "read" out using speech synthesis); allowing export of PDF files in RTF format, so that PDF is now no longer a "cul-de-sac"; and enabling text reflow so that PDF can be viewed on hand-held devices and WAP phones.

If one creates PDF using either FrameMaker 6 or the new macros for Word 2000, then any text styles are taken into the PDF as tags, which remain associated with the content in any subsequent export. If PDF is created in other ways, there may be no tags. And certainly a PDF created under Acrobat 3 or Acrobat 4 will have no tags. However, one can download from the Adobe Web site a plug-in for Acrobat 5.0 that is called "Make accessible." This analyzes the PDF and "guesses" the structure. For straightforward documents it generally gets it right. However, with complex documents, particularly if they have been scanned in, there can be problems that need to be sorted out manually. Manual adjustment of the tags, which is of course possible, is fine for a small number of documents. However, if one has a large number of documents, a test is advisable, because even a small percentage can involve many hours of hand-fixing.

PostScript and thus earlier versions of PDF have no concept of document structure; they essentially describe where to put an image, be it a picture or a string of characters, on a page. PostScript is, after all, a page description language. Adobe is now saying that PDF has broken away from PostScript. Previously, for example, if one selected and copied text on a PDF page, then there was a good chance that columnar text would come out in an incorrect order. This, of course, means that it is useless for speech synthesis, for export as RTF, and for reflowing to fit on a small screen. Now, however, after a file has been "made accessible" (i.e., tagged), Acrobat knows what the order of the text is. It has to be admitted that the tagging is fairly basic, with a distinct similarity to HTML, but such simplicity will not necessarily be important, and certainly not for speech synthesis or reflowing. Another aspect of tagging and making accessible is that different types of components can be handled in different ways, so that, for example, tables are not represented in WAP/PDA viewing, while images can be replaced with descriptive text for speech synthesis. This begins to touch on the topic of metadata, which is discussed in the next section.

In summary, tagged PDF does not use XML per se, although an export to PDF from FrameMaker+SGML that has an XML structure will carry that structure through to an export (e.g., in a form). However, this is a big step by Adobe, and it is clear that future versions of Acrobat and PDF will move even closer to structured documents, particularly as there is a possibility that FrameMaker and FrameMaker+SGML will merge in the next release. If that actually happens, then it seems likely that the integration with PDF will become even closer.

Metadata

There was metadata in PDF 1.3, stored in a dictionary. However, version 1.4 introduces a different model of how the metadata is stored, so that metadata can be associated either with the entire document or with individual components. There are two main reasons for this. One is to make metadata for PDF documents more easily available to Web search engines, while the other relates to one of the main uses of PDF, mainly in the printing industry, where metadata is associated with graphics in a PDF-based workflow. The new model means that metadata can be accessed by software later in the process.

The metadata will be represented in a format based on XML and RDF, to be defined in a new document entitled Adobe *XAP Metadata Framework*. *XAP* stands for eXtensible Authoring and Publishing Metadata Framework and is intended to be adopted more widely than just by applications that process PDF. XAP includes a method to embed XML data within non-XML data files in a platform-independent format that is easy to locate by scanning the document files, rather than parsing them.

While the XAP document is not yet available,

there is some information in *Portable Document Format: Changes from Version1.3 to1.4*, and Chuck Myers (http://www2.gca.org/knowledgetechnologies/2001/proceedings/Myers%20Slides.pdf) gives a considerable amount of detail as well as some sample XAP metadata. This presentation also gives indications of the direction in which Adobe is moving. XML is definitely an important item on Adobe's agenda.

David Penfold is a writer, editor, and publishing consultant (Edgerton Publishing Services). Trained as a physicist, he began his publishing career in academic journals with the International Union of Crystallography and then moved to the world of typesetting, working on many early disk and database conversion projects, including the Revised English Bible, for the Charlesworth Group in Yorkshire. He was project manager for the Electronic Journals on SuperJanet (SuperJournal) project in 1993 and has advised many U.K. publishers on production, content management, and implementing SGML. He is currently chairman of the British Computer Society (BCS) Electronic Publishing Specialist Group and a member of the BCS Publications Board. He is also an evaluator and reviewer for the European Commission Directorate General Interactive Electronic Publishing. His publications include several books and numerous articles, mainly concerned with electronic and conventional publishing and related computing topics.

XSLT Design Patterns

Jeni Tennison
mail@jenitennison.com
Jeni Tennison Consulting Ltd
4 Dudley Court, Beeston, Nottingham NG9 3HZ, United Kingdom
www.jenitennison.com/consulting

As with any programming language, using XSLT involves more than knowing about what constructs can be used. Most real-world applications require convoluted transformations that can be achieved only by bringing together XPath functions and XSLT elements and attributes in a complex structure. In addition, stylesheets need to be extensible, maintainable, and reusable. This paper discusses the design patterns that can help the XSLT author to create XSLT applications, ranging from low-level programming idioms, through transformation essentials such as recursion and looping and sorting and grouping, to higher-level issues such as general processing methods and combining stylesheets.

Introduction

XSLT is a programming language that's specifically designed to help you access, query, and manipulate XML. It's not a general programming language; it's designed for one thing: to take in XML and output something else. If you have some XML and you want to change it into another vocabulary so that you can share it with a partner organization, or format it so that you can display it on the Web, or analyze it to create a bar chart, then XSLT is the language for you.

Now that's not to say that XSLT is perfect, even at doing what it's designed to do. Real applications have a vast variety of requirements, and there's no way that such a young language, even built on all those years of experience with DSSSL, can meet them all. However, just like any other programming language, XSLT is flexible enough that you can usually bend it to do what you want it to do; what's more, you can often do it in several different ways.

The purpose of this paper is to go through some of the common things that people want to do in XSLT and talk about the best way to do them. Unfortunately, there isn't enough space here to go through all the alternatives or to give lots of examples. Nevertheless, the problems that I describe here are ones that crop up repeatedly on XSL-List (http://www.mulberrytech.com/xsl/xsl-list) and have been discussed and used several times. The solutions

are designed to score highly on three criteria:

efficiency—how quickly the code will be run by an XSLT processor,
readability—how easy it is for someone to read and understand the XSLT code, and
maintainability—how easy it will be to change and extend the code in the future.

You'll notice that many of the design patterns in this paper have some strange name associated with them. Sometimes they're hard to remember, and often they're hard to know how to pronounce, but it's good to recall the inspired and inspiring people who came up with shortcuts and workarounds for the rest of us.

I'm going to start with the small design patterns: the programming idioms that can be used in XPath and XSLT. Then I'll move up a notch to talk about recursion and iteration, controlling the flow within XSLT stylesheets. Following that, I'll cover some of the larger problems to do with sorting and grouping. Finally, I'll talk about some common stylesheets that can be used as the basis of greater things.

Programming Idioms

In this section, I'll cover some of the programming idioms that can be useful in constructing XPaths or in doing some of the smaller bits of code.

The Kaysian Method: Testing Identity

Testing the identity of a node against another node is an essential component of quite a few techniques in XSLT, most notably the Muenchian Method. One way of doing this is to test whether the unique identities of two nodes are the same, using either an identifying attribute (what attribute you use depends on the XML source vocabulary) or the unique ID generated by the `generate-id()` function.

```
generate-id( node1 ) = generate-id(node2 )
```

The Kaysian Method uses the fact that node sets never hold more than one copy of the same node. If a node set is made up of a union between two nodes, and both nodes are the same node, then the resulting node set will contain only one node. So, if counting the nodes in the node set gives you the value `1`, then the nodes are the same.

```
count( node1 |node2 ) = 1
```

Extending this, the same method can be used to test if a node is part of a set. If it is, then the union of the node with the set will result in a node set that is the same length as the original node set.

```
count( node |node-set ) = count(node-set )
```

Similarly, you can work out whether a node set is a subset of another set of nodes. If it is, then the union of the subset with the superset will result in a node set that is the same length as the original superset.

```
count( subset |superset ) = count(superset )
```

Namespace-Independent Attribute Matching

Testing whether an element is *not* named something is relatively easy to do with XPath:

```
not(self:: namespace-prefix :element-name )
```

However, the `self::` axis doesn't give access to attributes. Lack of attribute access is irritating because using XPath node tests to check the names of nodes involves some very clever namespace matching behind the scenes, and it would be nice to take advantage of it.

One possibility is to use `local-name()` and `namespace-uri()` to identify whether the attribute belongs to a particular namespace:

```
not(namespace-uri() = namespace-uri and
  local-name() = attribute-name )
```

There's nothing wrong with this, but having to write out the URI of the namespace is a little ugly.

Fortunately, you can take advantage of the fact that each element can have only one attribute with a particular name. It's easy to identify the attribute called that (if there is one), and then test whether that attribute is the same as the one you're looking at. You can do this in two ways, using `generate-id()`:

```
generate-id() != generate-id(../@name
  space-prefix :attribute-name )
```

or using the Kaysian Method described above:

```
count(.|../@ namespace-prefix :attribute-
name ) != count(../@namespace-
prefix :attribute-name )
```

The Becker Method: Getting Conditional Strings

One of the more frustrating limitations of XPath is that it doesn't support an if function that allows you to choose what string or number to return based on a boolean expression. Getting a boolean value is obviously very easy; getting different node sets is relatively easy:

```
node-set-if-true   [boolean-test  ] |
  node-set-if-false   [not( boolean-test  )]
```

However, to define a variable that can hold one of two strings depending on a boolean condition, then the normal pattern is

```
<xsl:variable name="variable-name  ">
    <xsl:choose>
        <xsl:when test="boolean-test  ">
          value-if-true      </xsl:when>
        <xsl:otherwise>
          value-if-false     </xsl:otherwise>
    </xsl:choose>
</xsl:variable>
```

Not only is this fairly verbose, it actually creates a result tree fragment variable rather than a string variable.

Getting around this isn't pretty, but you can do so by taking advantage of the `substring()`. The `substring()` function takes three arguments: the string, the first character of the substring you want, and (optionally) the number of characters in the string (if it isn't specified, then you get the rest of the string).

If the second argument is infinity, then none of the characters in the string is selected, whereas if the second argument is one (and there's no third argument), then all the characters are selected. One way of making infinity is to divide a number (say `1`) by `zero` (whereas you can get one by dividing `1` by `1`). When interpreted as a number, `true` is given the value `1` and `false` is given the value `0`.

The pattern is thus the following:

```
concat(substring(   value-if-true  ,
       1 div boolean-test  ),
       substring(  value-if-false  ,
       1 div not(boolean-test  )))
```

You can use this concatenation of strings to get numbers as well as strings conditionally.

The Allouche Method: Managing Whitespace

One of the difficulties when creating an XSLT stylesheet is getting it to put whitespace where you want whitespace and not put whitespace where you don't want whitespace. If you type some text content, then any whitespace around that text gets added to the result, with the result that indents in your stylesheet get carried over to your output no matter how much you adjust the `indent` of your `xsl:output` element.

To prevent spurious whitespace being added, you can use the `xsl:text` element, which essentially brackets off the text that you want to add to the output from the whitespace that you're using to make it look pretty in the stylesheet:

```
<xsl:text>  text </xsl:text>
```

Sometimes, using `xsl:text` can get very laborious. For example, adding brackets around the value of the current node without adding unnecessary whitespace involves:

```
<xsl:text>(</xsl:text>
<xsl:value-of select="." />
<xsl:text>)</xsl:text>
```

Often, this bracketing can be alleviated by using the `concat()` function to concatenate the text and the relevant expressions together:

```
<xsl:value-of select=
  "concat('(', ., ')')" />
```

Sometimes, however, it can't. In these cases, Allouche's Method comes into its own. In Allouche's method, an empty `xsl:text` element is used to separate unnecessary whitespace from the text that you want. If there is only whitespace between it and the preceding element, then that whitespace is ignored; if there is only whitespace between it and the *following* element, then *that* whitespace is ignored.

```
<xsl:text />
  (<xsl:value-of select="." />)
<xsl:text />
```

Recursion and Iteration

In this section I'll talk about design patterns for controlling flow in XSLT. There's a danger here: because XSLT is a declarative language, people with backgrounds in procedural languages often make it hard

to start thinking in a way that helps them to write XSLT to do what they want it to do. They may think that they want a loop to iterate a number of times, but actually want to collect that number of nodes and process them as a group. However, there are obviously cases where the equivalent of a procedural method is much easier to work with.

The Piez Method: Iterating a Number of Times

It can often be useful to add the same thing to your output multiple times. In cases involving repeated copies of a character, you can use the `substring()` on a string consisting of that character repeated many times. The following, for example, allows you to specify the number of tabs to be included:

```
substring('  &#x9;&#x9;&#x9;&#x9;&#x9;&#x9;
    &#x9;&#x9;&#x9;&#x9    ;', 1, times )
```

The above method works only for strings, and best for characters. An alternative method has to be used for elements or other result tree fragments. One method is to define a recursive template that takes the number of repetitions as a parameter. It outputs the content and if the number of repetitions is more than one, calls itself with the counter dimished by one:

```
<xsl:template name="repeat">
   <xsl:param name="count" select="1" />
   <xsl:param name="content" />
   <xsl:copy-of select="$content" />
   <xsl:if test="counter > 1">
      <xsl:call-template name="repeat">
         <xsl:with-param name="count"
           select="$count - 1" />
      </xsl:call-template>
   </xsl:if>
</xsl:template>
```

This template is called with the `$count` parameter set to the number of times the content should be repeated:

```
<xsl:call-template name="repeat">
   <xsl:with-param name="count"
      select=" times " />
   <xsl:with-param name="content">
      content     </xsl:with-param>
</xsl:call-template>
```

One disadvantage of this method is that it can be deeply recursive, which may cause problems with some XSLT processors. Another irritation is that calling the template is quite verbose.

The Piez Method uses the `xsl:for-each` element to iterate a number of times. The `xsl:for-each` element selects a number of nodes and then runs once for each of those nodes. If the right number of nodes are selected, then the content runs the right number of times.

The nodes that are used in the Piez Method can come from anywhere: they are never used in the content of the `xsl:for-each` and so don't have to be a particular type or have a particular value. One good source of nodes is the stylesheet itself. If you are going to use this technique a lot, it might be worth setting up a global variable to hold them:

```
<xsl:variable name="lots-of-nodes"
  select="document('')//node()" />
```

Once this has been set up, the design pattern for looping a number of times with an `xsl:for-each` element is

```
<xsl:for-each select="$lots-of-nodes
[position() &lt;= times ]"> content
</xsl:for-each>
```

There are two things to watch out for when you use this method. The first is that you have to make sure that the number of nodes in the set you're using as a basis for the iteration (the `$lots-of-nodes` variable, for example) is greater than the maximum number of times you'll ever want to repeat the content. The second is that within the `xsl:for-each`, the current node changes; if the content that you want repeated relies on values from the current node, then you will need to set up variables to use them within the `xsl:for-each`.

Finding a Maximum

The problem of finding the maximum value for an expression is a good example of a range of problems that involve working out a single value from a number of values held in a node set. Finding out the maximum node value for a set of nodes is relatively easy:

```
node-set [not( node-set > .)]
```

However, this solution will get vastly more inefficient the more nodes there are in the node set. It also doesn't work if the value that you want to find the maximum for is calculated based on a node rather than simply being its string value.

Another method to get the maximum is to sort the nodes in descending order based on the calculated

value that you're after, then pick off the first one and return that:

```
<xsl:for-each select="node-set ">
   <xsl:sort select="expression  "
      order="descending" />
   <xsl:if test="position() = 1">
      <xsl:value-of select=" expression  " />
   </xsl:if>
</xsl:for-each>
```

The problem with this method is, again, that it involves testing the calculated value for each of the nodes against the calculated value for a number of other nodes. The more nodes you have, the less efficient this gets.

The alternative solution that gets around this problem is to step through the nodes one by one. There are several recursive solutions that can be used; the best limits the depth of the recursion while ensuring that each node is visited once and only once. The recursive template at the heart of this method is a noded template that works out the value for the current node, then checks to find the next node that has a larger value. If there is a node with a larger value, then it applies templates to that node; if not, then it returns the value for the current node.

```
<xsl:template match="node()" mode="maximum">
   <xsl:variable name="value"
      select=" expression  " />
   <xsl:variable name="next"
      select="following-sibling::node()
      [expression   > $value][1]" />
   <xsl:choose>
      <xsl:when test="$next">
         <xsl:apply-templates
           select="$next" mode="maximum" />
      </xsl:when>
      <xsl:otherwise>
         <xsl:value-of select="$value" />
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

To use this method, the nodes must all be related to each other so that it's easy to step from one to the next. Setting this up involves creating a result-tree fragment that acts as a new document: this facility is available only in XSLT 1.1 or with extension functions that turn result-tree fragments into node sets that are available in most XSLT processors.

```
<xsl:variable name="node-set">
   <xsl:copy-of select="node-set " />
</xsl:variable>
```

With this node set in place, the maximum can be found by applying templates to the first node in the node list in the `maximum` mode:

```
<xsl:apply-templates select="$node-set[1]"
  mode="maximum" />
```

Sorting and Grouping

Sorting and grouping nodes in different ways are the more common complex requirements from XSLT.

Sorting in Arbitrary Order

XSLT provides the `xsl:sort` element for sorting nodes when iterating over them with `xsl:for-each` or applying templates to them with `xsl:apply-templates`. The `xsl:sort` element allows you to sort either alphabetically or numerically, in ascending or descending order. However, it doesn't let you sort in arbitrary order such as `first, second, third` or `January, February, March`.

The first step to sorting in arbitary order is to define that order. This involves making a list of nodes, in the order you want them in, with the values that you have in your source XML. For example, you need to set up months with something like this:

```
<month>January</month>
<month>February</month>
<month>March</month>
<month>April</month>
...
```

These months need to be made accessible as a node set through a variable, either by defining them within a variable in XSLT 1.1 (or with an appropriate extension function) or by defining them in the stylesheet or another document and accessing them through the `document()` function. For the sake of this example, let's say that I've defined them within a `$months` variable.

Now, given a month name, I can retrieve its position among its peers using

```
count($months[. = current(@month)]/
  preceding-sibling::month)
```

Sorting on this position will put the nodes I'm sorting in the same, arbitrary, order as I've defined:

```
<xsl:for-each select="$dates">
   <xsl:sort select="count($months
```

```
    [. = current(@month)]/preceding-
    sibling::month)" data-type="number" />
   ...
</xsl:for-each>
```

Grouping by Position

There's often a requirement in XML+XSLT applications, especially those creating HTML, to show only part of the data at a time. You might want to let people page through tables of information, for example. These subsets of data are based on the number of items that they contain.

The first step is to identify the first item in the group. Often in paging applications, this is passed into the stylesheet as a parameter:

```
<xsl:param name="show" select="1" />
```

The first item in the only group to be shown can then be identified with the expression:

```
node-set [number($show)]
```

In applications where several groups are displayed at the same time, such as having the headers of a table repeat every 20 rows, then you have to pick out those items that are first in each group according to their position. If you want groups of size 20, for example, then the first items in each group are the 1st, 21st, 41st (and so on) nodes. These nodes can be picked out with the expression:

```
node-set [position() mod group-size = 1]
```

Once the first item in a group is identified, it's relatively easy to get the rest of the items in the group. If we have groups of 20 nodes, then starting from the first node, they will be the next 19 nodes. Usually the items for the group are siblings within the source XML, so the XPath for gathering the items is

```
. | following-sibling::item-node-test
  [position() &lt; group-size ]
```

Putting this together, a typical application for grouping by position applies templates to the first nodes in a group in `group` mode:

```
<xsl:apply-templates select="  item-node-test
  [position() mod group-size  =
  1]" mode="group" />
```

There is then a template matching the items in `group` mode, which gathers the group together and adds elements around it. To give the content of the group, it then applies templates to the items in the group in `item` mode:

```
<xsl:template match="item-node-test  "
mode="group">
    <grouping-element>       group-header
<xsl:apply-templates select=". | following-
  sibling:: item-node-test  [position() &lt;
  group-size ]" mode="item" />
  group-footer     </grouping-element>
</xsl:template>
```

Finally, there is a template that matches the items in `item` mode to give the output for each item within the group:

```
<xsl:template match="item-node-test  "
  mode="item"> item-content  </xsl:template>
```

The Muenchian Method: Grouping by Value

The Muenchian Method of grouping is one of the more well-known design patterns. It allows you to group items according to some attribute or element value, or even combinations of these values. It follows the same basic pattern as grouping by position, shown above, but also draws on methods of testing node identity and the `key()` function.

As with grouping by position, the Muenchian Method is divided into two stages: finding the first item in a group and identifying the rest of the items in the group. In the Muenchian Method, the groups are defined using a key to make this easier. The key is defined with the `xsl:key` element:

```
<xsl:key name="key-name " match="item-
  node-test " use="grouping-property  " />
```

A typical key definition for grouping `employee` elements by their manager attribute, for example, would be

```
<xsl:key name="employees-by-manager"
  match="employee" use="@manager" />
```

Once a key has been set up, the `key()` function can be used to retrieve all the items that have a particular value for the grouping property:

```
key( key-name , key-value )
```

For example, to find all the employees that are managed by `Wilma`, you can use

```
key('employees-by-manager', 'Wilma')
```

Given this expression, the first item in each group can be identified by the fact that it is the first item that is retrieved when the key is used with a particular key value. For example, if `Fred` is the first employee that's retrieved to with the above use of `key()`, then he should be first in the group of employees managed by `Wilma`. Testing whether an item is the first item in the group it belongs to involves testing its identity against the identity of the first item in the group; you can use the Kaysian Method for this:

```
count(. | key(key-name ,
  grouping-property  )[1]) = 1
```

Thus, to get the first items in each group, you can use the following expression:

```
item-node-test  [count(. | key(key-name ,
  grouping-property  )[1]) = 1]
```

The completed template is similar to the one above for grouping by position, but with different ways of selecting the first item in each group and the rest of the items in each group. First, apply templates to the first nodes in a group in group mode:

```
<xsl:apply-templates select="  item-node-test
  [count(. | key(key-name ,
  grouping-property  )[1]) = 1]"
  mode="group" />
```

Have a template matching the items in `group` mode, which gathers the group together and adds elements around it. To give the content of the group, apply templates to the items in the group in `item` mode:

```
<xsl:template match="item-node-test  "
mode="group">
    <grouping-element>        group-header
<xsl:apply-templates select="key(key-name ,
  grouping-property  )" mode="item" />
  group-footer    </grouping-element>
</xsl:template>
```

Finally, have a template that matches the items in item mode to give the output for each item within the group:

```
<xsl:template match="item-node-test  "
  mode="item">  item-content  </xsl:template>
```

Organizing Stylesheets

XSLT was originally developed as a means of restructuring and creating formatting objects for a piece of XML so that it could be presented in a human-readable way. XSLT is still used to transform XML into presentable formats like formatting objects, HTML and WML, but it's also being used as a generic means for transforming XML into other XML vocabularies in order to share information between applications.

There are thus four general types of stylesheets in use, depending on the type of XML vocabulary that's being transformed from and the type of document that it's being transformed to:

Data to Data—translating between XML vocabularies, filtering data, and typical B2B;
Data to Document—transforming for presentation and typical B2C;
Document to Data—document analysis; and
Document to Document—transforming for different presentation and filtering content.

When the input and output formats are similar to each other, such as in data-to-data and document-to-document transformations, then the structure of the input is usually the driving force behind the structure of the output. On the other hand, when they are very different, the structure desired for the output drives the way the input is processed.

This distinction is mirrored in the stylesheets: when the input drives the output, then *push* structures are used; when the output is the driving force, then *pull* structures are used. Push stylesheets involve a lot of applying templates, letting the XSLT processor and the structure of the input determine which templates should be applied when. Pull stylesheets involve picking bits of information from the input; this can still involve applying templates but is more likely to involve applying them to specific nodes or calling named templates.

In the rest of this section, I'll look at two examples. The first is the basic copying stylesheet, which works wholly on the push principle and can be used to filter out information from a document or some XML data. The second is a stylesheet that constructs a document based on some XML data, such as creating a letter based on customer information.

The Duplicating Stylesheet

The following stylesheet can be used to create a logical copy of the input XML:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/
  Transform">

<!-- copy all elements and their attributes
  and content recursively -->
<xsl:template match="*">
    <xsl:copy>
       <xsl:apply-templates select="@*" />
       <xsl:apply-templates />
    </xsl:copy>
</xsl:template>

<!-- copy all attributes, comments and pro
  cessing instructions -->
<xsl:template match="@*|comment()
  |processing-instruction()">
    <xsl:copy-of select="." />
</xsl:template>

</xsl:stylesheet>
```

Note that I said "logical copy." Running the above stylesheet will generate a document tree for the input document and then output a serialized version of a copy of this tree. This means that there might be changes such as the addition of namespace declarations, changes to the quotes used around attributes or even the addition of defaulted attributes from the XML DTD. These changes don't alter the logical content of the document, but they might alter how it looks.

This stylesheet can be used as the basis for various different stylesheets. It's ideal if you want to keep your document basically as it is but make just a few changes here and there. One stylesheet that I base on it is a reformatting stylesheet. I strip all the white-space-only text from the tree:

```
<xsl:strip-space elements="*" />
```

get rid of leading and trailing whitespace from text nodes:

```
<xsl:template match="text()">
    <xsl:value-of
     select="normalize-space()" />
</xsl:template>
```

and get the XSLT processor to output it nicely indented for me:

```
<xsl:output indent="yes" />
```

You can use it to filter out nodes that you don't want. For example, if the difficulty of a particular section were indicated through a `difficulty` attribute on each `section` element, then I could filter out all the difficult sections by adding the following template:

```
<xsl:template match="section
  [@difficulty = 'difficult']" />
```

You can also use it to add nodes. For example, to add `id` attributes to every element to hold a unique ID for each, I could change the element-matching template in the coyping stylesheet to

```
<xsl:template match="*">
    <xsl:copy>
       <xsl:attribute name="id">
       <xsl:value-of
         select="generate-id()" />
       </xsl:attribute>
       <xsl:apply-templates select="@*" />
       <xsl:apply-templates />
    </xsl:copy>
</xsl:template>
```

The Stylesheet Template

In some cases, you may be interested in simply picking information out of the input XML to insert into an output document. The design that XSLT provides for this is the "Literal Result Element as Stylesheet" stylesheet. With this type of stylesheet, the user can write a stylesheet as if she or he is writing the output document, with a few XSLT instructions indicating where content from the input should be included.

```
<html xsl:version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/
     Transform">
    <head>
       <title>Statement for Account
       <xsl:value-of select="Statement/
         @AccountNum" /></title>
    </head>
    <body>
       <h1>Account <xsl:value-of
       select="Statement/@AccountNum" /></h1>
       <p>For <xsl:value-of select=
         "Statement/AccountHolder" /></p>
       <hr />
       <xsl:for-each
         select="Statement/Transaction">
           ...
       </xsl:for-each>
       <hr />
       <p class="disclaimer">
           ...
       </p>
```

```
    </body>
</html>
```

There are two disadvantages to this approach. One is that most people who design pages for the Web or write formal letters don't want to have to learn XSLT and XPath. The people who write the applications need to allow them to write templates without involving any weird syntax. The second is that when a literal result element is used as the stylesheet document element, you're very restricted in the XSLT that you can use: you can't include templates of any kind, you can't import anything, you can't specify the output method, and so on.

Because of these restrictions, a common application design is to allow designers to create template documents that hold the desired structure for the output while programmers write XSLT stylesheets that take these template documents, along with some XML data, and process them together to produce the desired output. The designers use special tags within the template to indicate where in the page information from the XML data should be entered. In essence, this turns the pull method used in the above stylesheet into a push method instead: the document template is used to drive the output.

The easiest way to manage this setup is to use two namespaces within the template document: one for the literal output and one for the places where information from the XML data should be inserted. For example, with the above you might have:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:data="http://www.bank.co.uk/
      accountdata">
   <head>
      <title>Statement for Account
        <data:AccountNum /></title>
   </head>
   <body>
      <h1>Account <data:AccountNum /></h1>
      <p>For <data:AccountHolder /></p>
      <hr />
      <data:Transactions />
      <hr />
      <p class="disclaimer">
         ...
      </p>
   </body>
</html>
```

The stylesheet needs to run on either the XML data or the template; it doesn't particularly matter which, although as the identity of the other file should either be hard coded or passed as a parameter into the stylesheet, it makes sense to use the more changeable of the two as the source. It needs to run over the template, copying elements in the output namespace and defining templates to be run on the elements in the data namespace that pick the relevant information from the XML data and process it. A basic stylesheet might look like this:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform"
    xmlns:out="  output-namespace-URI    "
    xmlns:ref="  data-reference-name
    space-URI  "
    xmlns:data="  data-namespace-URI    " />

<!-- root node of the template document -->
<xsl:variable name="template"
  select="document('   template.xml  ')" />
<!-- root node of the <abbrev>XML
  </ abbrev> data -->
<xsl:variable name="data" select="/" />

<!-- process the template document -->
<xsl:template match="/">
   <xsl:apply-templates
    select="$template/*" />
</xsl:template>

<!-- copy elements in the output
  namespace -->
<xsl:template match="out:*">
   <xsl:copy>
      <xsl:apply-templates select="@*" />
      <xsl:apply-templates />
   </xsl:copy>
</xsl:template>

<!-- copy text, attributes in the output
  namespace, and any unprefixed attributes
  on elements in the output namespace -->
<xsl:template match="text() | @out:* |
  out:*/@*">
   <xsl:copy-of select="." />
</xsl:template>

<!-- process elements that are references
  to the <abbrev>XML</abbrev> data by
  retrieving that data -->
<xsl:template match="ref:      reference-element   ">
value-of-relevant-data    </xsl:template>

...
</xsl:stylesheet>
```

Combining Multiple Stylesheets

When XSLT authors have several stylesheets that work on the same data, they often start to wonder about having a single stylesheet that takes a parameter and decides, according to that parameter, which of the stylesheets to use to transform the data. The pattern that they come up with for this is to conditionally include the stylesheet that they want based on the parameter. Unfortunately, that doesn't work: XSLT doesn't support conditional inclusion of stylesheets.

So how do you do it? One way is to get a scripting language to decide which stylesheet to use, but I'm focusing on XSLT solutions here. The XSLT solution is to have a single central stylesheet, import all the stylesheets that you want to combine into it, and then use nodes to only use the templates from one of them.

The first step is to identify any templates that are used in all the stylesheets that you have and put them either within the central stylesheet or within a new utility stylesheet that is imported into all the others. The latter option is better because it means that the individual stylesheets can still work indpendently.

The second step is to add a node to each of the unmoded templates within the stylesheets that you're combining. The nodes have to be different for each stylesheet so that the templates don't clash; using the stylesheet name might be one way of ensuring that. Any `xsl:apply-templates` instructions that don't include a node should have this node specified as well.

The third step is to add a root-node-matching template to each of the stylesheets that will let you run them individually. This template should apply templates to the root node in the stylesheet mode:

```
<xsl:template match="/">
   <xsl:apply-templates select="."
    mode=" stylesheet-mode   " />
</xsl:template>
```

Once that's added, you should be able to run each of the stylesheets on its own as if nothing has changed.

The final step is to construct the central stylesheet. This stylesheet should simply import all the other stylesheets, define the parameter(s) that identify which stylesheet should be used, and have a single template that directs the processing to the relevant stylesheet. Note that you have to use an `xsl:choose` for this because the mode attribute on `xsl:apply-templates` is not an attribute value template.

```
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/
      Transform" />

<!-- import stylesheets -->
<xsl:import href="stylesheet1.xsl  " />
<xsl:import href="stylesheet2.xsl  " />
...

<!-- define parameter -->
<xsl:param name="processing-mode" />

<!-- direct processing -->
<xsl:template match="/">
   <xsl:choose>
      <xsl:when test="$processing-mode =
           'mode1 ">
         <xsl:apply-templates select="."
           mode=" stylesheet1-mode   " />
      </xsl:when>
      <xsl:when test="$processing-mode =
        'mode2 ">
         <xsl:apply-templates select="."
           mode=" stylesheet2-mode   " />
      </xsl:when>
      ...
      <xsl:otherwise>
         <xsl:apply-templates select="."
           mode=" default-mode  " />
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

Conclusions

In this paper, I've gone through some of the design patterns that have been developed for XSLT. Naturally, this paper doesn't describe all the things that you want to do with XSLT, or all the ways of solving the problems that are discussed here. But I hope it has given you a taste of some of the guidelines that have been developed for its use.

XSLT is still quite a young language and, with XSLT 1.1 on the horizon and countless extension elements and functions being tried and tested, a constantly evolving one. As XSLT grows, it may well come to include easier ways to tackle some of the problems addressed above in a simpler fashion than the methods I've described here. I hope that these design patterns do *not* stand the test of time: there should be easier ways of accomplishing the same things, and, as XSLT continues to be developed, no doubt there will be.

One of the great features of XSLT is its extensibili-

ty. Already, we've seen the power of that with the various XSLT processors adopting their own extension elements and functions, some of which have been incorporated into XSLT 1.1. With XSLT 1.1 and the `xsl:script` element, we, as users, will be able to add to this spirit of extension, testing and evolution.

No doubt, as XSLT grows and evolves, we will find new things that we want to do with it, and new design patterns will gradually emerge for doing them. If you discover one, I hope you'll share it so that we as a community can evolve along with XSLT.

My thanks to those who shared the design patterns that I've described above—Mike Kay, Oliver Becker, David Allouche, Wendell Piez, and Steve Muench—as well as the other fine folks on XSL-List who have contributed greatly to their discussion and documentation, especially Dave Pawson, David Carlisle, Dimitre Novatchev, and Mike Brown.

Jeni Tennison is an independent consultant specializing in XML+XSLT application design and development, technical authoring, tutoring and mentoring of XML and XSLT, and training course development and delivery in XSLT, data modeling, and XML Schemas. She was previously a senior knowledge engineer and security controller with Epistemics Ltd. She holds the Ph.D. in psychology and artificial intelligence from the University of Nottingham, where her research focused on Web communities and knowledge engineering, including the development of a MOO-based HTTP server that formed the basis of a collaborative knowledge server.

Calendar

If you know of any events that we should include in this listing, please let us know about it through the contact details at the front of *interChange*.

September

17–20 September 2001
XML World 2001: The XML Technologies Cornerstone Event
San Francisco, California, U.S.A.
Contact: rwellige@xmlworld.org,
http://www.xml-world.org/xmlsan01/index.htm

19–21 September 2001
XML One : Amsterdam
Amsterdam, The Netherlands
Contact: www.xmlone-amsterdam.com

XML One: Amsterdam will provide an intensive three-day, three-track technical program offering intuitive, neutral, and focused training.

24–25 September 2001
Integrating XML and EDI
San Francisco, California, U.S.A.
Contact : www.oasis-open.org/events/index.shtml

This two-day conference offers information on facilitating business processes, increasing operational efficiency, and strengthening trading relationships using RosettaNet Standards.

30 September–4 October 2001
XML One & Web Services One
San Jose, California, U.S.A.
Doubletree Hotel
Contact : www.xmlconference.com/sanjose

The West Coast's premier XML training event is co-locating with the new Web Services conference. Attendees may register for either conference or both.

The conferences run 30 September–4 October, with exhibits 1–2 October.

October

8 –11 October 2001
XML DevCon—Europe—Fall
Olympia Conference Centre, London, U.K.
Contact : http://www.camelot.com

Presented by overwhelming demand, this will be the most sophisticated XML event to come to London in 2001.

22–25 October 2001
XML Edge 2001
Santa Clara, California, U.S.A.
Contact: http://www.oasis-open.org/events/index.shtml

This conference program will offer information on XML and related technologies, XML repositories and schemas, XSL/XSLT/XNTML, JAXP/DOM2/SAX, XML and technologies, XML and databases, real-world case studies, and XML applications.

November

5–8 November 2001
XML Asia Pacific
Sydney, Australia
Contact : www.oasis-open.org/events/index.shtml

This year's conference shows how new XML improvements over the last 12 months have taken the language from theoretically important to practical and essential to the growth of the information sector. Industry's key figures will present their insights into how XML is emerging, how it can be applied, and how to be a part of the process.

20–21 November 2001
Forum XML & e-Business Integration
Paris, France
Contact : http://www.technoforum.fr/
Pages/forumXXML01/index.html

This event is dedicated to XML for the French market. Among the topics to be covered are Web Services, Enterprise Application Integration (EAI) and application servers, Corporate Portals architectures and Content management, EDI and ebXML, Supply-Chain Integration, E-marketplaces, B2B platforms, component-based architectures, XML and J2EE, and XML and data modeling.

27–30 November 2001
2nd International Workshop on Conceptual Modeling Approaches for e-Business
Yokohama, Japan
Contact : www.oasis-open.org/events/index.shtml

This workshop is being held in conjunction with the 20th International Conference on Conceptual Modeling. The scope of the workshop is very much in sync with what OASIS is doing with the United Nations CEFACT in ebXML. Please submit all topics and/or questions to Professor Heinrich Mayr (mayr@ifit.uni-klu.ac.at).

December

9–14 December 2001
XML 2001
Walt Disney World Dolphin Hotel
Orlando, Florida, U.S.A.
Contact : http://www.gca.org/attend/2001_conferences/xml_2001/

This conference will feature top-quality tutorials and talks that will help attendees understand what works now and what doesn't, as well as provide glimpses into the future. The attendees at XML 2001 will hear firsthand from the experts which products will solve their problems and which methodologies will meet their needs.

International SGML/XML Users' Group Bookstore List

Please note that we are closing out the Bookstore. If you would like to order any of the following books, now is the time!

Your order can be placed by post to the ISUG Bookstore, Copse House, 15 Upton Close, Swindon, Wiltshire, SN25 4UL. UK, or by telephone or fax +44 (0) 1793 721106 or email: yvonne@isug.freeserve.co.uk

Cheques should be made payable to the International SGML/XML Users' Group, or supply an official order or order reference if you wish your organization to be invoiced.

Clearance Sale: ALL books at half-price while stocks last!

Publication Title	List Price	Copies	Member Price	Item Total
ISO 8879 SGML (includes Amendment 1)	£140.00		£60.00	
ISO 8879 SGML Amendment 1 only	£9.00		£3.00	
ISO 9069 SGML Document *Interchange* Format (SDIF)	£31.00		£15.00	
ISO/TR 9544 Vocabulary of Computer-assisted Publishing	£81.00		£36.00	
ISO/TR 9573 Techniques for Using SGML	£131.00		£50.00	
ISO 10744 HyTime	£131.00		£50.00	
ISO 12083 Information and documentation —Electronic manuscript preparation and markup	£147.00		£65.00	
ISO 10744 Annex B Supplementary Materials (ask for list)	N/A		£14.00	
L. Alschuler: *ABCD . . . SGML A Users' Guide to Structured Information*	£29.50		£14.75	
N. Bradley: *The Concise SGML Companion*	£19.95		£9.95	
M. Bryan: *An Author's Guide to SGML*	£27.50		£13.75	
M. Colby, D.S. Jackson: *Special Edition Using SGML*	£46.99		£23.50	
S.J. DeRose/D.G.Durand: *Making Hypermedia Work*	£59.50		£29.75	
S.J. DeRose: *The SGML FAQ Book*	£46.25		£23.25	
T. Donovan: *Industrial Strength SGML: An Introduction to Enterprise Publishing*	£26.95		£13.50	

B. DuCharme: *SGML CD*	£39.99	£19.95
C. Ensign: *$GML, the Billion Dollar Secret*	£18.95	£9.50
C.F. Goldfarb: *The SGML Handbook*	£50.00	£25.00
C.F. Goldfarb, S. Pepper, C. Ensign: *SGML Buyer's Guide*	£43.99	£21.95
R.A. Jelliffe: *The SGML Cookbook B/CD*	£43.75	£21.95
E. Maler & J. El Andaloussi: *Developing SGML DTDs from Text to Model to Markup*	£39.95	£19.95
S. McGrath: *Parseme 1st: SGML for Software Developers*	£26.95	£13.50
Y. Rubinsky, M. Maloney: *SGML on the Web*	£28.99	£14.50
J.M. Smith: *SGML and Related Standards*	£31.50	£15.75
R.C. Turner, T.A. Douglas & A.J. Turner: *Readme 1st: SGML for Writers and Editors*	£29.95	£14.95
B. von Hagen: *SGML for Dummies*	£28.99	£14.50
E. van Herwijnen: *Practical SGML, 2nd edition*	£56.95	£28.50
SoftQuad, SGML Primer	£14.00	£7.00
Bulletin Back Issues: Volumes 1, 2, 3, 4 (price per volume)	£18.00	£12.00
Newsletter Back Issues: Volume 1, Issues 1–10, 11–20, 21–31 (price per set)	£25.00	£20.00
Volume 2, Issues 1–4 (set)	£20.00	£15.00
Volume 3, Issues 1–4 (set)	£20.00	£15.00
Volume 4, Issues 1–4 (set)	£20.00	£15.00
Volume 5, Issues 1–4 (set)	£20.00	£15.00
Postage & Packing £2.50 (each book)		
Total of Order		

Clearance Sale: ALL books at half-price while stocks last!